

# ASYN/StreamDevice Support Frameworks

Eric Norum



# ASYN

- What is it?
- What does it do?
- How does it do it?
- How do I use it?

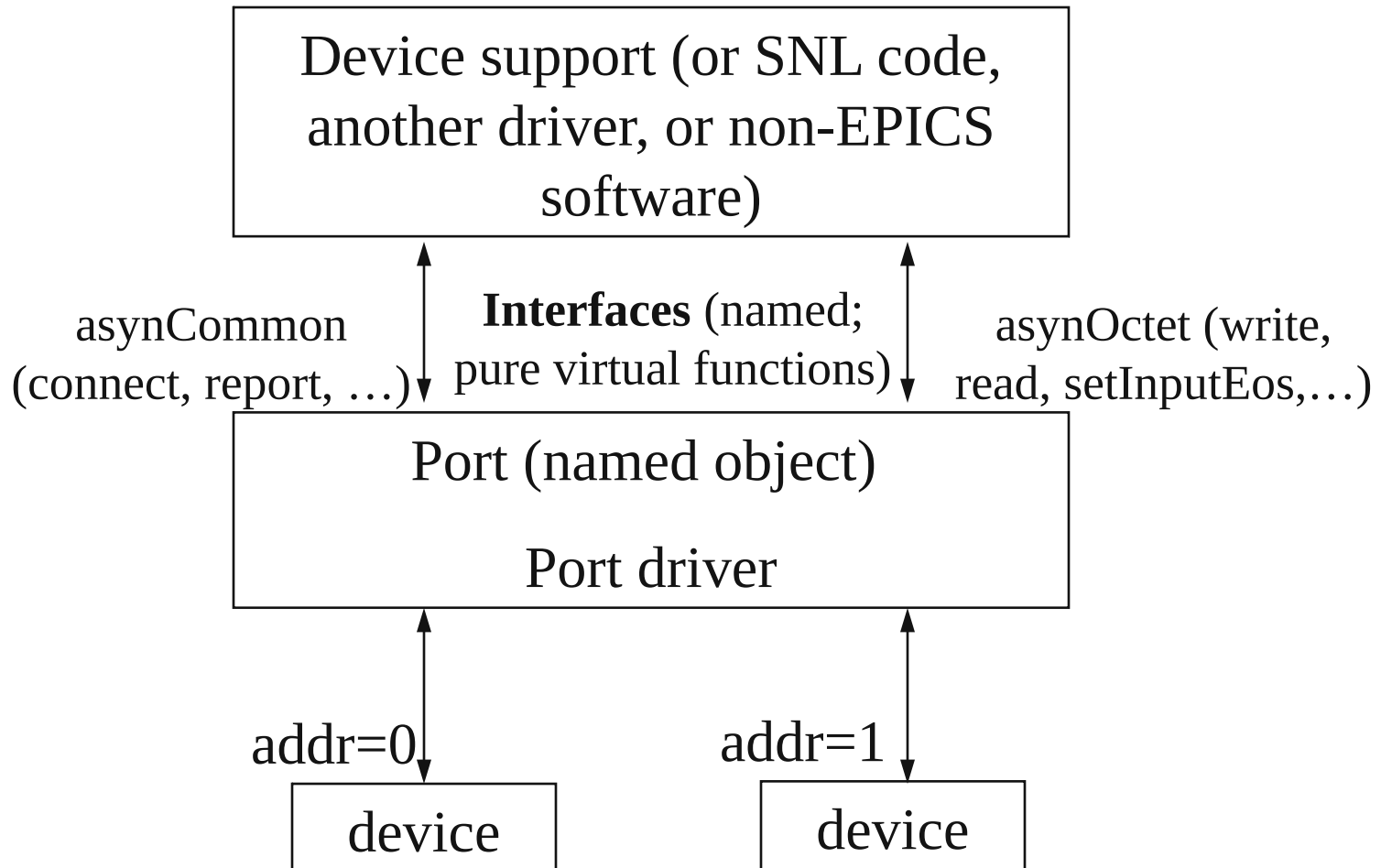


# What is it?

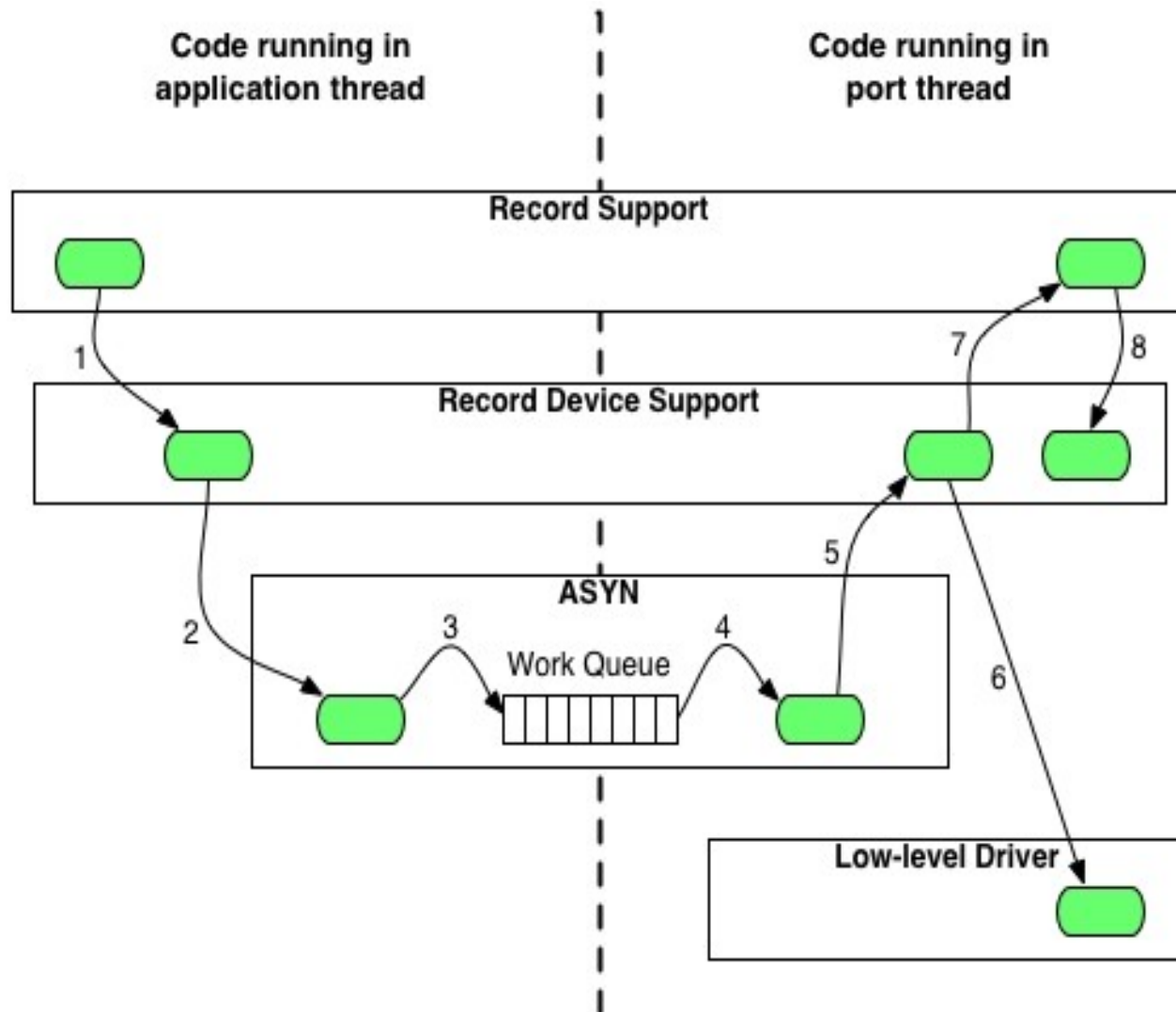
Asynchronous Driver Support is a general purpose facility for interfacing device specific code to low level communication drivers



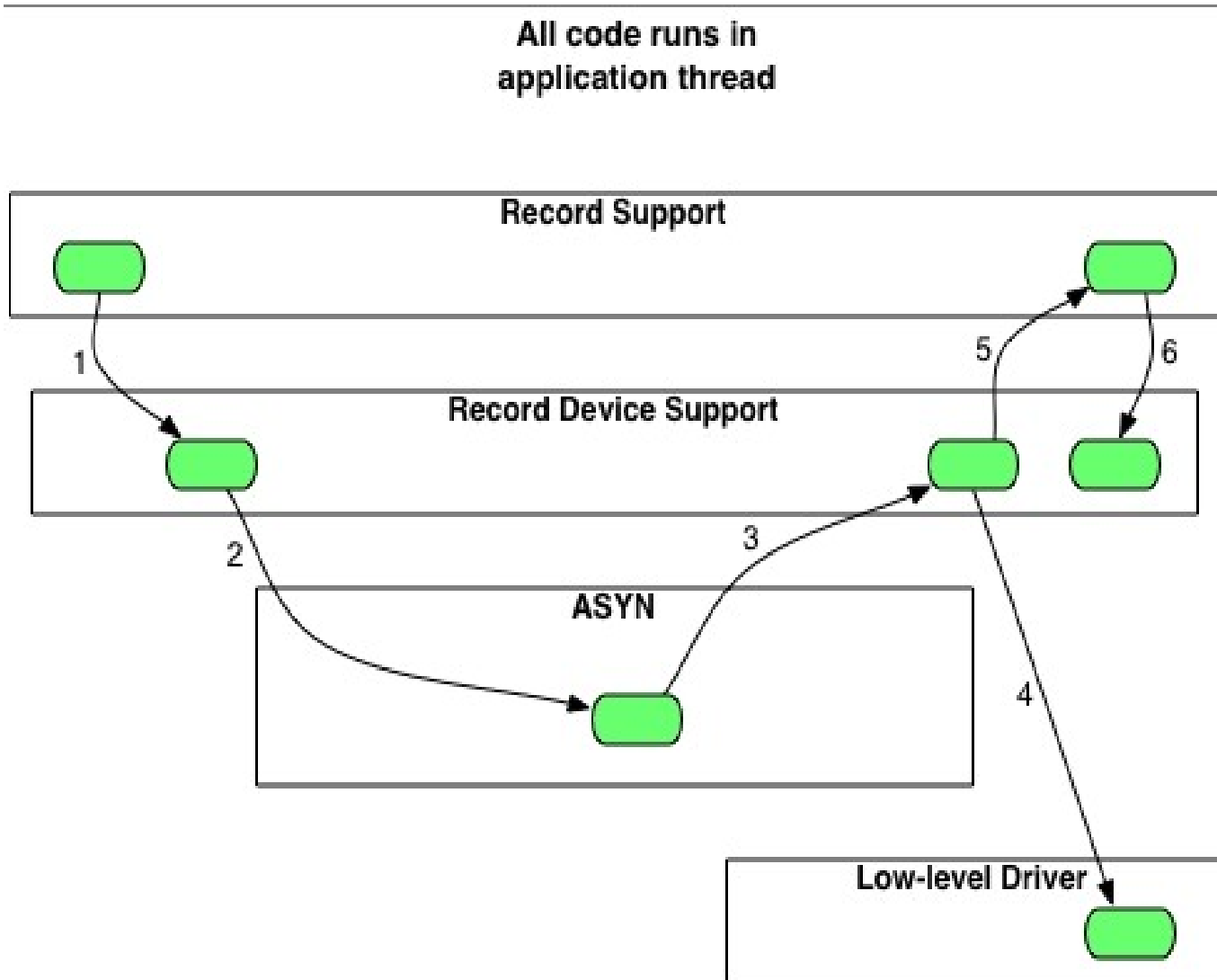
# asyn Architecture



# Control flow – asynchronous driver



# Control flow – synchronous driver



# ASYN Components – asynManager

- Provides thread for each communication interface
  - All driver code executes in the context of this thread
- Provides connection management
  - Driver code reports connect/disconnect events
- Queues requests for work
  - Nonblocking – can be called by scan tasks
  - User-supplied callback code run in worker-thread context makes calls to driver
  - Driver code executes in a single-threaded synchronous environment
- Handles registration
  - Low level drivers register themselves
  - Can ‘interpose’ processing layers



# ASYN Components – asynCommon

- A group of methods provided by all drivers:
  - Report
  - Connect
  - Disconnect
  - Set option
  - Get option
    - Options are defined by low-level drivers
    - e.g., serial port rate, parity, stop bits, handshaking





# ASYN Components – asynOctet

- Driver or interposed processing layer
- Methods provided in addition to those of asynCommon:
  - Read
  - Write
  - Set end-of-string character(s)
  - Get end-of-string character(s)
- All that's needed for serial ports, 'telnet-style' TCP/IP devices, USB-TMC.
- The single-threaded synchronous environment makes driver development much easier
  - No fussing with mutexes
  - No need to set up I/O worker threads



# ASYN Components – asynGpib

- Methods provided in addition to those of asynOctet:
  - Send addressed command string to device
  - Send universal command string
  - Pulse IFC line
  - Set state of REN line
  - Report state of SRQ line
  - Begin/end serial poll operation
- Interface includes asynCommon and asynOctet methods
  - Device support that uses read/write requests can use asynOctet drivers. Single device support source works with serial or GPIB.



# ASYN Components – asynRecord

- Diagnostics
  - Set device support and driver diagnostic message masks
  - No more ad-hoc ‘debug’ variables!
- General-purpose I/O
  - Replaces synApps serial record and GPIB record
- Provides much of the old ‘GI’ functionality
  - Type in command, view reply
  - Works with **all** asyn drivers
- A single record instance provides access to all devices in IOC



# asynRecord

- EPICS record that provides access to most features of asyn, including standard I/O interfaces
- Applications:
  - Control tracing (debugging)
  - Connection management
  - Perform interactive I/O
- Very useful for testing, debugging, and actual I/O in many cases
- **If your IOC uses ASYN it should provide at least one asynRecord to give clients control of diagnostic messages!**

The screenshot shows a graphical control panel for the 'asynRecord siocbcm:asyn'. At the top, the title bar reads 'asynRecord siocbcm:asyn'. Below it, a blue header bar contains the text 'siocbcm:asyn'. The main interface includes several input fields and buttons. The 'Port' field is set to 'BCM' and the 'Address' field is set to '-1'. A green 'Connect' button is present. Below these, the 'drvInfo' field is empty and the 'Reason' field is set to '0'. The 'Interface' dropdown menu is set to 'asynOctet'. There are two buttons at the bottom of this section: 'Cancel queueRequest' and 'More...'. A section labeled 'Error:' is currently empty. Below this, there are three status indicators: 'Connected' (green), 'Enabled' (green), and 'autoConnect' (blue). Each indicator has a corresponding button: 'Connect', 'Enable', and 'autoConnect'. The bottom section contains two columns of checkboxes for tracing. The left column is labeled 'traceMask' and has a value of '0x1'. The right column is labeled 'traceIOMask' and has a value of '0x0'. The left column includes checkboxes for 'traceError', 'traceIODevice', 'traceIOFilter', 'traceIODriver', 'traceFlow', and 'traceWarning'. The right column includes checkboxes for 'traceIOASCII', 'traceIOEscape', 'traceIOHex', 'Truncate size' (set to 80), 'traceInfoMask' (set to '0x1'), 'traceInfoTime', 'traceInfoPort', 'traceInfoSource', and 'traceInfoThread'. At the bottom, there is a 'Trace file' field set to 'Unknown'.

# asynRecord – asynOctet devices

## Interactive I/O to serial device



asynOctet.adl

13LAB:serial7

Timeout (sec): 1.0000 Transfer: Write/Read

asynOctet interface: Supported Active

**Output** Format: ASCII Terminator: \r

ASCII: tptptp

Length: Requested: 80 Actual: 6

**Input** Format: ASCII Terminator: \r

ASCII: 1TP30.001,2TP0.000,3TP-0.001,4TP0.000

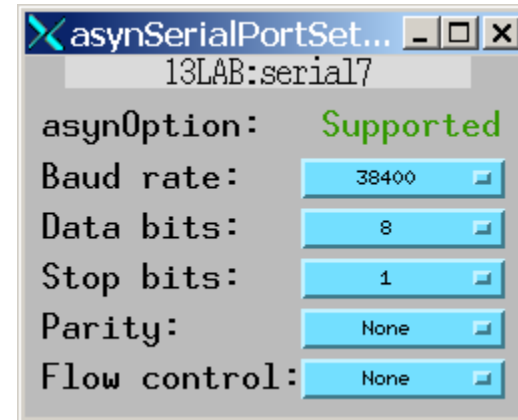
Length: Requested: 0 Actual: 37

EOM reason: Eos

I/O Status: NO\_ALARM I/O Severity: NO\_ALARM

Scan: Passive Process More...

## Configure serial port parameters



asynSerialPortSet...

13LAB:serial7

asynOption: Supported

Baud rate: 38400

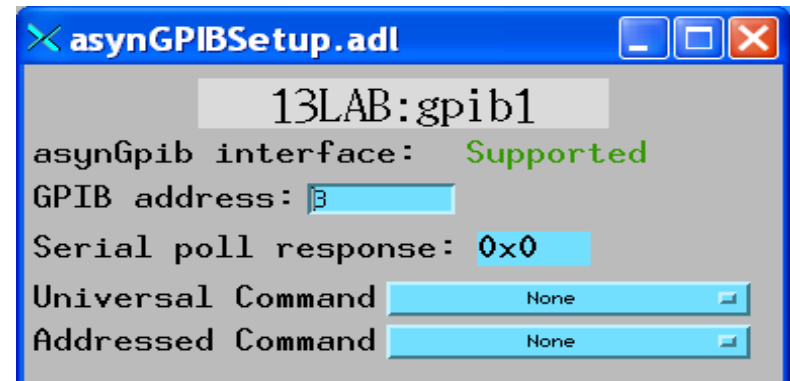
Data bits: 8

Stop bits: 1

Parity: None

Flow control: None

## Perform GPIB-specific operations



asynGPIBSetup.adl

13LAB:gpi1

asynGpib interface: Supported

GPIB address: 3

Serial poll response: 0x0

Universal Command: None

Addressed Command: None

# asynRecord – register devices

Same asynRecord, change to ADC port

**asynRecord.adl**

13LAB:serial7

Port:  Address:

**Connected**

drvInfo:  Reason:

Interface:

Error:

**Connected** **Enabled** **autoConnect**

traceMask		traceIOMask	
<input type="text" value="0x1"/>	<input type="text" value="0x0"/>	<input type="text" value="0x0"/>	<input type="text" value="0x0"/>
<input type="checkbox"/> Off <input type="checkbox"/> On	<input type="checkbox"/> Off <input type="checkbox"/> On	<input type="checkbox"/> Off <input type="checkbox"/> On	<input type="checkbox"/> Off <input type="checkbox"/> On
<input type="checkbox"/> traceError	<input type="checkbox"/> traceIOASCII	<input type="checkbox"/> traceIODevice	<input type="checkbox"/> traceIOEscape
<input type="checkbox"/> traceIOFilter	<input type="checkbox"/> traceIOHex	<input type="checkbox"/> traceIODriver	<input type="text" value="80"/> Truncate size
<input type="checkbox"/> traceFlow			
Trace file: <input type="text" value="Unknown"/>			

Read ADC at 10Hz with asynInt32 interface

**asynRegister.adl**

13LAB:serial7

Timeout (sec):  Transfer:

Interface:	Int32	UInt32Digital	Float64
<input type="text" value="asynInt32"/>	<b>Supported</b>	<b>Unsupported</b>	<b>Supported</b>
	<b>Active</b>	<b>Inactive</b>	<b>Inactive</b>

Output:

Output (hex):

Input:

Input (hex):

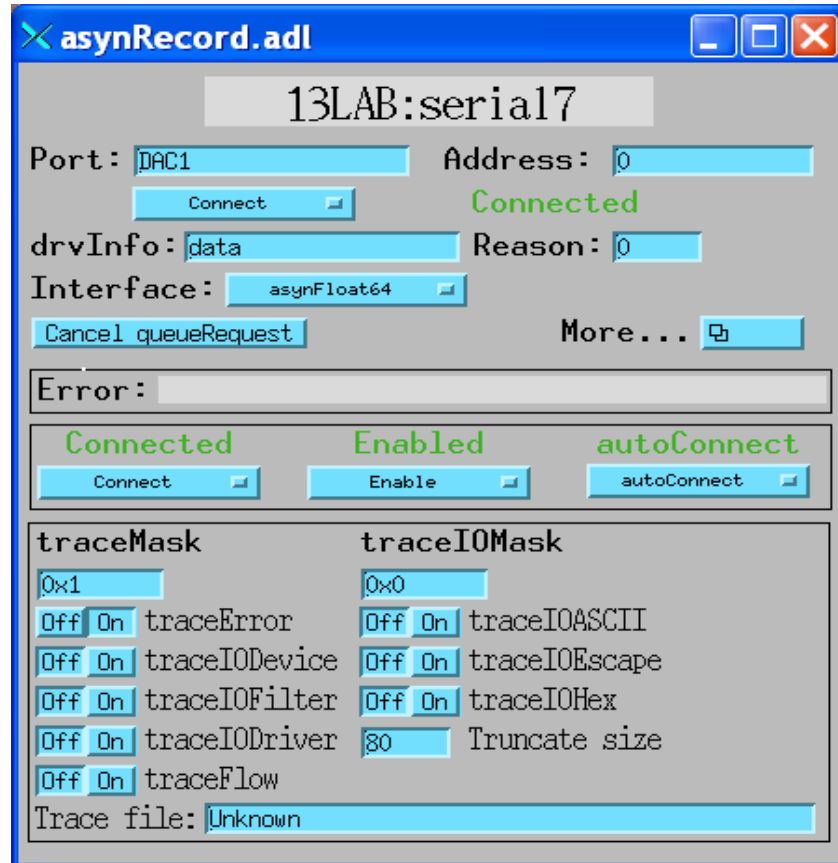
Mask (hex):

I/O Status: **NO\_ALARM** I/O Severity: **NO\_ALARM**

Scan:

# asynRecord – register devices

Same asynRecord, change to DAC port



The **asynRecord.adl** window shows the configuration for the device **13LAB:serial7**. The **Port** is set to **DAC1** and the **Address** is **0**. The **Interface** is **asynFloat64**. The status is **Connected**. The **drvInfo** is **data** and the **Reason** is **0**. The **Trace Mask** and **Trace IOMask** are both **0x1**. The **Trace file** is **Unknown**.

Port: **DAC1** Address: **0**

Connect **Connected**

drvInfo: **data** Reason: **0**

Interface: **asynFloat64**

Cancel queueRequest More...

Error:

**Connected** **Enabled** **autoConnect**

Connect Enable autoConnect

traceMask traceIOMask

**0x1** **0x0**

Off On traceError Off On traceIOASCII

Off On traceIODevice Off On traceIOEscape

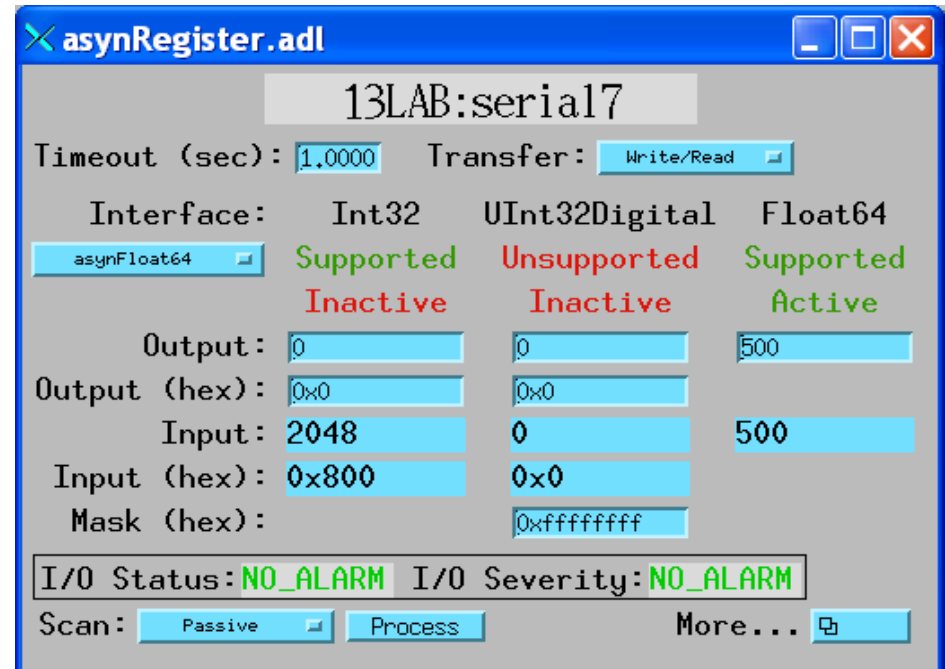
Off On traceIOFilter Off On traceIOHex

Off On traceIODriver **80** Truncate size

Off On traceFlow

Trace file: **Unknown**

Write DAC with asynFloat64 interface



The **asynRegister.adl** window shows the configuration for the device **13LAB:serial7**. The **Timeout (sec)** is **1.0000** and the **Transfer** is **Write/Read**. The **Interface** is **asynFloat64**. The **Output** is **0** and the **Input** is **2048**. The **Mask (hex)** is **0xffffffff**. The **I/O Status** is **NO\_ALARM** and the **I/O Severity** is **NO\_ALARM**. The **Scan** is **Passive**.

Timeout (sec): **1.0000** Transfer: **Write/Read**

Interface: **asynFloat64** **Supported** **Unsupported** **Supported**

**Inactive** **Inactive** **Active**

Output: **0** **0** **500**

Output (hex): **0x0** **0x0**

Input: **2048** **0** **500**

Input (hex): **0x800** **0x0**

Mask (hex): **0xffffffff**

I/O Status: **NO\_ALARM** I/O Severity: **NO\_ALARM**

Scan: **Passive** **Process** More...

# Tracing and Debugging

- Standard mechanism for printing diagnostic messages in device support and drivers
- Messages written using EPICS logging facility, can be sent to stdout, stderr, or to a file
- Device support and drivers call:
  - `asynPrint(pasynUser, reason, format, ...)`
  - `asynPrintIO(pasynUser, reason, buffer, len, format, ...)`
  - Reason:
    - `ASYN_TRACE_ERROR`
    - `ASYN_TRACEIO_DEVICE`
    - `ASYN_TRACEIO_FILTER`
    - `ASYN_TRACEIO_DRIVER`
    - `ASYN_TRACE_FLOW`
    - `ASYN_TRACE_WARNING`
- Tracing is enabled/disabled for (port/addr)
- Trace messages can be turned on/off from iocsh, vxWorks shell, and from CA clients such as EDM via `asynRecord`
- `asynOctet` I/O from shell

The screenshot shows the 'asynRecord siocbcm:asyn' configuration window. At the top, the title bar says 'asynRecord siocbcm:asyn'. Below it, the main title is 'siocbcm: asyn'. There are input fields for 'Port' (set to 'BCM\_CMD') and 'Address' (set to '-1'). A 'Connect' button is present. Below these, 'drvInfo' is empty and 'Reason' is set to '0'. The 'Interface' is set to 'asynOctet'. There are buttons for 'Cancel queueRequest' and 'More...'. An 'Error:' section is empty. Below that, there are three status indicators: 'Connected', 'Enabled', and 'noAutoConnect', each with a corresponding button. The bottom section contains two columns of settings: 'traceMask' and 'traceIOMask'. Each has a hex value field and a list of options with 'Off/On' toggle buttons. The 'Trace file:' field at the bottom is set to 'Unknown'.





# Typical source file arrangement

- Instrument support is placed in  
*.../modules/instrument/<instrumentname>/Rx.y/*
- Each *<instrumentname>/Rx.y/* directory contains at least

*Makefile*

*configure/*

*<InstrumentName>Sup/*

*documentation/*

*License*



# Script to make this a little easier

- `mkdir /.../modules/instrument/myinst/head`
- `cd /.../modules/instrument/myinst/head`
- `/<path to ASYN support module>/bin/<arch>/makeSupport.pl  
-t streamSCPI myinst`

`Makefile`

`configure/...`

`myinstSup/`

`Makefile devmyinst.db devmyinst.proto`

`documentation/`

`devmyinst.html`

- A few changes to the latter 3 files and you're done!
- Notice that there are no C or C++ files.
  - Running `make` just copies the `.db` and `.proto` files to the support module top-level `db/` directory.



# Introduction to Stream Device

- Generic EPICS device support for devices with “byte stream” communication.
  - RS-232 (Local serial port or LAN/Serial adapter)
  - TCP/IP
  - VXI-11
  - GPIB (Local interface or LAN/GPIB adapter)
  - USB-TMC (Test and Measurement Class)
- A single stream device module can serve to communicate using any of the above communication mechanisms.



# Introduction to Stream Device

- Command/reply messages:
  - \*IDN?
  - SET:VOLT 1.2
  - Non-ASCII 'strings' too
- Command generation and reply parsing configured by ***protocols***
- Formatting and interpretation handled with ***format converters***
  - Similar to C printf/scanf format strings
  - Custom converters too, but not easy



# Stream Device *Protocols*

- Defined in *protocol files*
- Plain ASCII text file
- No compiling – IOC reads and interprets protocol file(s) at startup
- Protocols are linear
  - No looping
  - No conditionals
  - Rudimentary exception handlers
- A single entry can read/write multiple fields in one or many records
- Output records can be initialized from instrument at IOC startup
  - With one big caveat – instrument must be on and communicating at IOC startup



# StreamDevice EPICS Database

```
record(bo, "$(P)$(R)CLS") {  
    field(DESC, "SCPI Clear status")  
    field(DTYP, "stream")  
    field(OUT, "@devmyInst.proto cmd(*CLS) $(PORT) $(A) ")  
}  
  
record(longin, "$(P)$(R)GetSTB") {  
    field(DESC, "SCPI get status byte")  
    field(DTYP, "stream")  
    field(INP, "@devmyInst.proto getD(*STB) $(PORT) $(A) ")  
}
```

- DTYP=stream
- INP/OUT fields specify protocol file name, protocol entry (with optional arguments), ASYN port and address.
- Address can be any value (typically 0) for single-address interfaces.



# StreamDevice Protocol File

```
cmd {  
    out "\$1";  
}  
getD {  
    out "\$1?";  
    in "%d";  
}
```

- Protocol entries contain statements to produce output and request input
- C-style escape sequence can be used ('\r', '\n', '\033', '\e')
- Format converters are similar to those used by C printf/scanf
  - By default the VAL or RVAL field is used as the data source/destination
  - Can refer to any field, even in another record



# StreamDevice Additional Records

DTYP  $\neq$  stream for protocol entry additional records:

```
record(stringin, "$(P)$(R)Serial")
{
    field(DESC, "Serial number")
    field(DTYP, "Soft Channel")
}
record(ai, "$(P)$(R)VP5")
{
    field(DESC, "+5V supply")
    field(DTYP, "Raw Soft Channel")
    field(EGU, "V")
    field(PREC, "3")
    field(LINR, "SLOPE")
    field(ESLO, "1e-3")
...
record(longin, "$(P)$(R)Temp1")
{
    field(DESC, "Sensor 1 temperature")
    field(DTYP, "Soft Channel")
}
```





# StreamDevice Protocol File

Protocol entries can be long – Use multiple lines and string concatenation to improve readability

```
query {  
    out "Q";  
    in ":"  
        "SN=% (\$1Serial.VAL) 39[^, ], "  
        "UN=% (\$1Name.VAL) 39[^, ], "  
        "IP=%*[^, ], "  
        "V3=%d, "  
        "V5=% (\$1VP5.RVAL) d, "  
        "V+12=% (\$1VP12.RVAL) d, "  
        "V-12=% (\$1VM12.RVAL) d, "  
        "T1=% (\$1Temp1.VAL) d, "  
        ...  
        "POH=% (\$1HoursOn.VAL) g, "  
        "MAXTMP=% (\$1MaxTemp.VAL) g; "  
}
```

Notice the use of the width field – guard against buffer overruns!



# StreamDevice Protocol File – Terminators

- Terminators can be set globally or per entry.
- Some interfaces can handle only a single character. If device replies with '\r\n' then specify *InTerminator*=' \n ' and ignore the '\r' in the reply.

```
InTerminator = "\n";  
OutTerminator = "\r";
```



# StreamDevice Protocol File – Initial Readback

- Useful to set initial value of output records to match the value presently in the instrument.
- @init 'exception handler'
- Often the same as the corresponding readback prototype entry

```
getF {  
    out "\$1?";  
    in "%f";  
}  
setF {  
    @init { out "\$1?"; in "%f"; }  
    out "\$1 %f";  
}
```

```
record(ao, "$ (P) $ (R) IntegrationTime")  
{  
    field(DESC, "Reading integration time")  
    field(DTYP, "stream")  
    field(OUT, "@devKeithley6487.proto setF(NPLC) $ (PORT) $ (A) ")
```



# Adding StreamDevice/ASYN instrument support to an application

- This is easy because the instrument support developers always follow all the guidelines – right?
- Most of these steps apply to pretty much any support module, not just StreamDevice/ASYN instruments.



# Make some changes to configure/RELEASE

- Edit the configure/RELEASE file created by makeBaseApp.pl
- Confirm that the EPICS\_BASE path is correct
- Add entries for the instruments and ASYN:

```
DAWN_RUSH =/usr/local/epics/R3.14.12/modules/instrument/DawnRuSH/R1-0  
ASYN      =/usr/local/epics/R3.14.12/modules/soft/asyn/asynR4-21  
EPICS_BASE=/home/EPICS/base
```



# Modify the application Makefile

```
!  
xxx_DBD += base.dbd  
xxx_DBD += stream.dbd  
xxx_DBD += drvAsynIPPort.dbd  
           (and/or drvAsynSerialPort.dbd, drvAsynUSBTMC.dbd, etc.)  
xxx_DBD += asyn.dbd  
!  
xxx_LIBS += stream asyn
```



# Modify the application database Makefile

Copy the instrument support database and prototype files to the application <top>/db/ directory:

*:*

```
DB_INSTALLS += $(DAWN_RUSH)/db/devDawnRuSH.db  
DB_INSTALLS += $(DAWN_RUSH)/db/devDawnRuSH.proto
```



# Modify the application startup script

```
epicsEnvSet ("CRATE_ADDRESS", "$ (CRATE_ADDRESS=crateapex01:23) ")
```

(above line is optional, but makes it easy to override for testing)

```
epicsEnvSet ("STREAM_PROTOCOL_PATH", "${TOP}/db")
```

```
;
```

```
drvAsynIPPortConfigure ("CR0", "$ (CRATE_ADDRESS) TCP", 0, 0, 0)
```

```
;
```

```
dbLoadRecords ("db/devDawnRuSH.db", "P=apexCrate:, R=1:, PORT=CR0")
```

- P,R – PV name prefixes – PV names are \$(P)\$ (R)name
- PORT– ASYN port name from corresponding **devxxxConfigure** command





# Lab Session

## Control 'network-attached device'

- Host `www.xxx.yyy.zzz` – TCP Port 24742
- '\n' command terminator, '\r\n' reply terminator
- \*IDN?
  - Returns device identification string (up to 100 characters)
- LOAD?
  - Returns three floating-point numbers separated by spaces (1, 5, 15 minute load average)
- ON?
  - Returns OFF/ON (0/1) status
- VOLTS?
  - Returns most recent voltage setting
- CURR?
  - Returns current readback ( $\pm 11A$ )



# Lab Session

## Control 'network-attached device'

- ON [0, 1]
  - Turns supply OFF/ON (0/1)
- VOLTS x.xxxx
  - Sets voltage ( $\pm 10V$  range)

